Fault tolerance

Introduction

Fault tolerance is **indispensable** to build dependable systems that have the following characteristics:

- **Availability**: the system needs to be ready for use as soon as it is needed
- **Reliability**: the system should be able to run continuously for a long time
- **Safety**: the system should not cause errors elsewhere
- **Maintainability**: the system should be easy to fix

> ✎ **Availability vs. Reliability**
>
> If a system goes down for one millisecond every hour, it has an availability of 99.999% but is still **highly unreliable**

From faults to failures

A system **fails** when it is **unable to provide its services**. A failure is the result of an **error**, which in turn is caused by a **fault**. Some faults can be avoided, while others cannot.

Building a dependable system demands the ability to **deal with the presence of faults**. A system is said to be **fault tolerant** if it can provide its services even in the presence of faults.

Faults can be classified according to the frequency at which they occur:

- **Transient** faults occur once and then disappear
- **Intermittent** faults appear and vanish with no apparent reason
- **Permanent** faults continue to exist until the failed components are repaired

There are also different types of failures:

- **Omission** failures:

    - In processes, we can have:
        - **Fail-safes**: when a fault occurs, the process outputs something wrong
        - **Fail-stops**: when a fault occurs, the process crashes in a detectable way
        - **Fail-silents**: when a fault occurs, the process crashes but the crash cannot be detected
    - In communication channels, we can have:
        - **Send omissions**
        - **Channel omissions**
        - **Receive omissions**
- **Timing** failures: occur when one of the time limits defined for the system is violated (only applies to synchronous systems)
- **Byzantine** (or *arbitrary*) failures:
    - Processes may omit intended processing steps or add unnecessary ones
    - Channels may send corrupted messages, deliver a message twice or deliver a message that does not exists

General techniques

The key technique to mask failures is **redundancy**, which can be of several types:

- **Information** redundancy
- **Time** redundancy
- **Physical** redundancy

An example: client/server communication

Reliable point-to-point communication is usually provided by the TCP protocol, which masks omission failures using **acknowledgements** (*acks*) and **retransmissions**.

Crash failures, however, like a connection loss, are not so easily masked. This becomes critical when trying to provide high-level communication facilities that completely mask communication issues. Masking communication problems completely is **just not feasible**.

Ideally, RPC should provide the semantics of procedure calls in a distributed setting, but in reality a number of problems arise:

- The server cannot be located

- Requests get lost
- The server crashes
- Replies get lost
- The client crashes

There are, fortunately, some "benign" cases:

- If the server cannot be located by the client, the client can handle this as an exception. This can also happen when the client is using an obsolete version of the protocol. This is a tolerable fault.
- If the request from the client is lost, the client can try and send it again. If too many requests get lost, the client can conclude the server is down.

Server crashes

A bigger problem arises when the **server** crashes: in this case, the client cannot tell when the server crashed. It is easy, however, to ensure that the server's job has been done at most once or at least once.

Let's take a print server as an example: this server performs two operations:

- Print the requested document (`P`)
- Send a confirmation message (`M`)

The message can be sent either before or after printing, but the server can crash at any time, of course.

Assuming that the client knows the server crashed, it can apply four strategies:

- Always reissue the same request
- Neve reissue the same request
- Reissue only when the confirmation message `M` was not received
- Reissue only when the confirmation message `M` was received

Some strategies may be better than other, but none of the ones above can result in "exactly once" semantics.

Similar problems to this can occur when the reply from the server is lost for some other reason. The server should cache clients' requests so as not to repeat duplicate

ones, using sequence numbers for example. The problem becomes how long to keep this information.

Client crashes

Of course, clients can crash as well. A computation started by a dead client is called **orphan**. Orphans still consume resources on the server, so the server must get rid of them.

There are several strategies to get rid of orphans:

- **Extermination**: RPCs are logged by the client and the orphans are killed on the client's request after a reboot. This is a costly procedure because it requires to log each call
- **Reincarnation**: when a client reboots, it starts a new epoch and sends a broadcast message to servers, who kill old computations started on behalf of that client. Replies sent by orphans have an obsolete epoch, so they are easily discarded.
- **Gentle reincarnation**: similar to reincarnation, but servers kill old computation only if their owner cannot be located.
- **Expiration**: remote computations expire after a while. Clients wait to reboot in order to let remote computations expire.

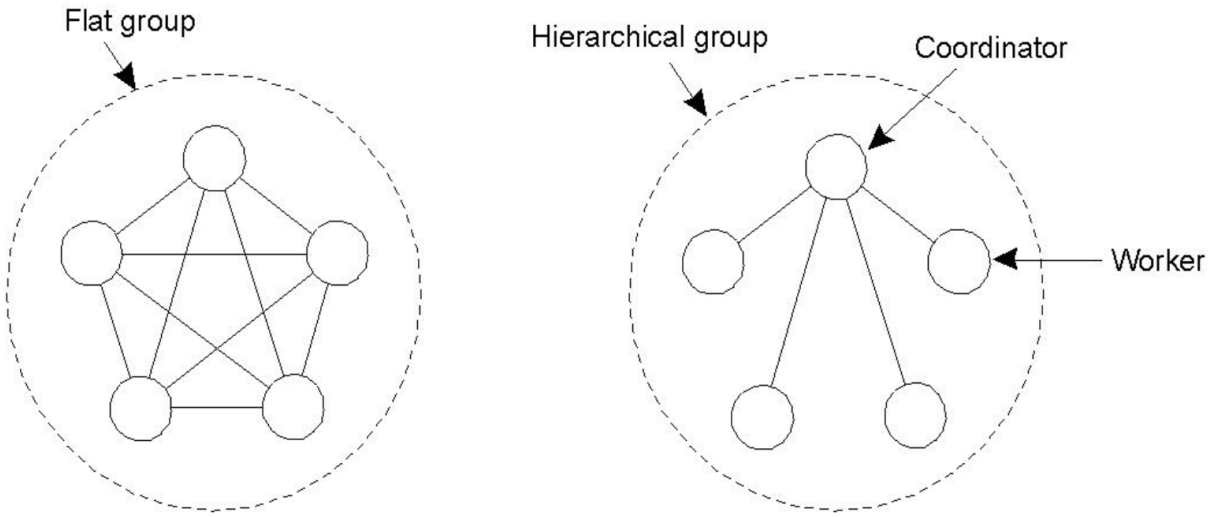
The reincarnation strategy seems to be the best, but what if a killed orphan was reading from some files? A lot of bookkeeping is needed to maintain a reincarnation strategy.

Protection against process failures

Process resilience

Redundancy can be used to mask the presence of faulty processes, with redundant process groups: if some processes fail, the healthy processes can continue to work. This is known as **process resilience**.

Process resilience can be implemented through two possible organisations: flat groups and hierarchical groups:



Using groups is easier said than done though, since this requires keeping track of which processes constitute each group.

Distributed membership management in flat groups requires that join and leave announcements are reliable and sent in multicast. The problem is that if a process crashes, it won't tell anyone that it is crashing. Furthermore, if too many processes in the same group crash, the group will have to be rebuilt.

How large should a group be?

Let's consider a replicated-write system where information is stored by a set of replicated processes:

- If processes fail silently, then $k + 1$ processes allow the system to be k-fault-tolerant
- If failures are Byzantine, matters become worse: $2k + 1$ processes are required to achieve k-fault tolerance (this is needed to have a working voting mechanism)

In practice we cannot be sure that no more than k processes will ever fail simultaneously, which is fine, but in some cases might not be enough.

Agreement in process groups

A number of tasks may require that the members of a group agree on some decision before continuing, for example to elect a coordinator or commit a transaction.

Since we are dealing with faults, we want all non-faulty processes to reach an agreement. This differs from the previous case, because the decision is not taken by an external observer.

If a set of processes must agree on some value, the value is subject to a validity condition: it cannot be just *any* value.

More precisely, we must make sure that these conditions are true:

- Each process starts with some initial value
- All non-faulty processes have to reach a decision based on these initial values
- The following properties must hold:
  - **Agreement**: no two processes decide on different values
  - **Validity**: if all processes start with the same value v, then v is the only possible decision value
  - **Termination**: all non-faulty processes eventually decide

The coordinated attack problem shows that consensus is not possible in the presence of arbitrary communication failures.

We will now analyse what happens when communication is reliable but processes are allowed to fail.

Let's review our assumptions:

- We consider a synchronous system in which all processes evolve in synchronous rounds
- We want our processes to reach agreement according to the previous definition of consensus

FloodSet algorithm

Luckily, this problem is easier than the previous one and can be solved through the **FloodSet algorithm**:

- Let v_0 be a pre-specified default value
- Each process maintains a set variable W initialised with its start value
- The following steps are repeated for $f + 1$ rounds
  - Each process sends W to all other processes
  - Each process adds the received sets to its instance of W

- The decision is made after $f + 1$ rounds:
  - If $|W| = 1$, then the value contained in W is selected
  - If $|W| > 1$, then the initial value v_0 is selected (or every process uses the same function to select another value, like $\max(W)$)

It is possible to prove that the FloodSet algorithm is **correct**.

It is also possible to reduce the complexity of this algorithm with a more optimised version: each process needs to know the entire set W only if its cardinality is greater than one, so the optimised algorithm broadcasts W only in the first round and each process in the group does so again only when a new value is discovered.

Introducing byzantine failures

This is fine, but introducing Byzantine failures makes matters more complex: now processes may exhibit arbitrary behaviours like sending arbitrary messages or performing arbitrary state transitions.

Our assumptions now change:

- **Agreement**: no two non-faulty processes decide on different values
- **Validity**: if all non-faulty processes start with the same value v, then v is the only possible decision value for all non-faulty processes
- **Termination**: all non-faulty processes eventually decide

The idea is that we cannot guarantee anything about faulty processes.

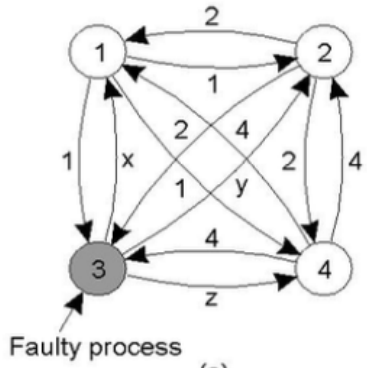
This new problem can be described as another formulation of the coordinated attack problem: imagine that n generals are ready to perform an attack and have to reach a coordinated decision on whether to attack or retreat. They announce their troop strengths and decide based on the total number of troops. Communication is perfect, but **some generals are traitors**. Is this problem even solvable in the first place?

Let's take a look at our assumptions for the Byzantine generals problem:

- **Agreement**: no two loyal generals learn different troop strength values
- **Validity**: if a loyal general announces v, then all loyal generals learn that his troop strength is v
- **Termination**: all loyal generals eventually learn the troop strength of all the other loyal generals

This is solvable through **Lamport's Algorithm**: let's suppose we have four generals and a traitor. We can follow these steps:

1. Send troop strength to others
2. Form a vector with received values
3. Send vector to others
4. Compute vector using the majority for each vector position



(a)

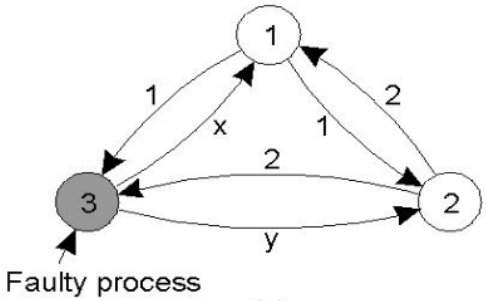
1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Lamport showed that if there are m traitors, $2m + 1$ loyal generals are needed for an agreement to be reached, for a total of $3m + 1$ generals. In the following example, we can see that there is not way for generals 1 and 2 to determine a vector which is both correct and equal to that computed by the others:



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

Considering asynchronous systems

So far, we only considered synchronous systems, but distributed systems are inherently asynchronous. Let's now consider the problem of reaching an agreement between n processes with 1 faulty process in an asynchronous system then.

Unfortunately, Fischer, Lynch and Paterson proved that **no solution exists** through the **FLP theorem**.

Even though this result was proved in the case of crash failures, since a Byzantine fault can simulate a crash fault, the solution for the Byzantine fault would also prove successful for crash faults.

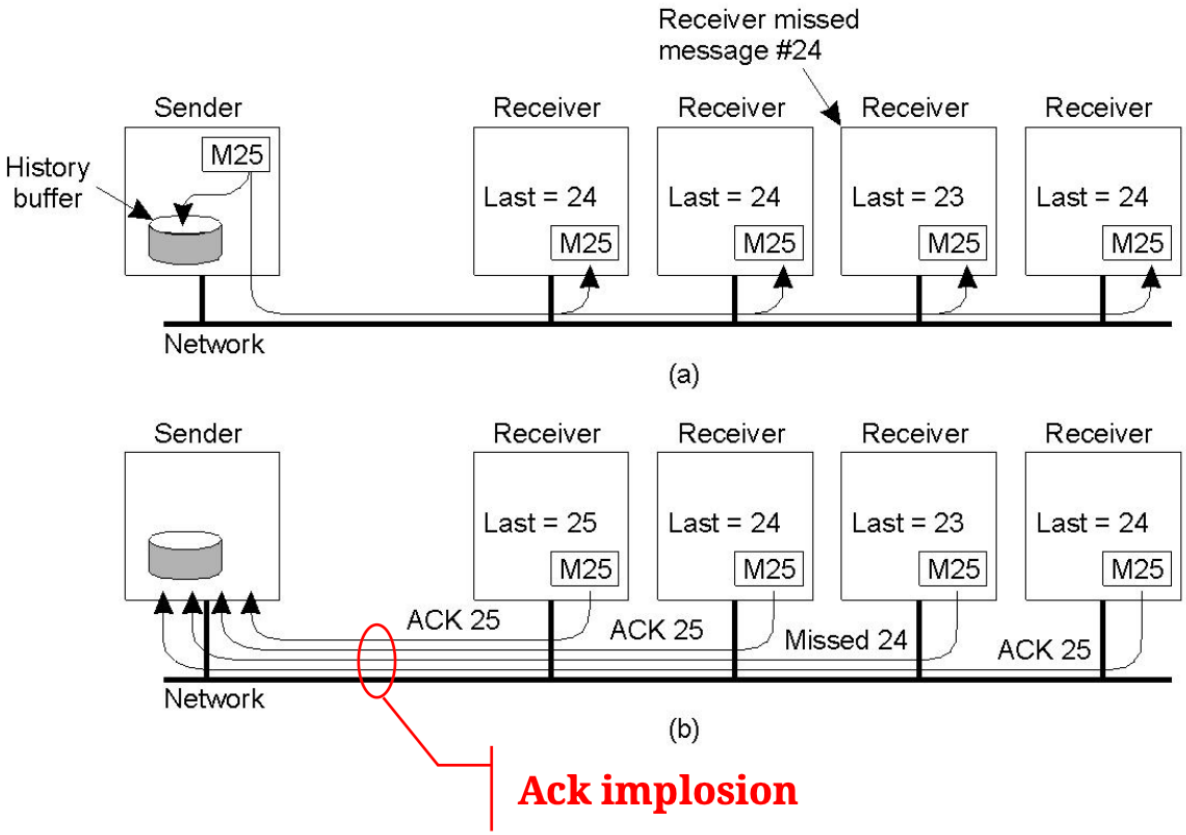
Reliable group communication

Reliable group communication is of critical importance if we want to exploit process resilience by replication. Achieving reliable multicast through multiple reliable point-to-point channels may not be efficient or sufficient.

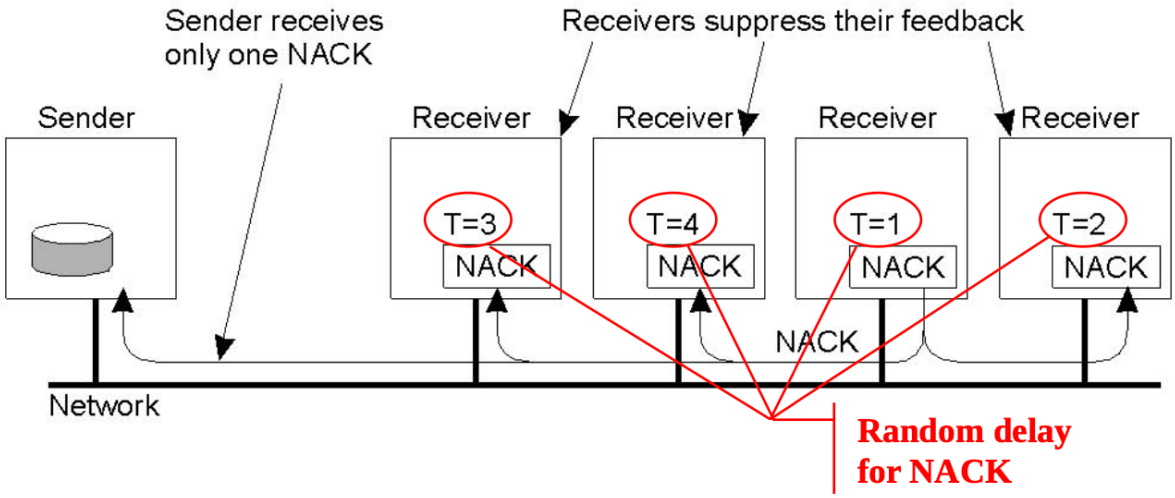
There are multiple scenarios we can analyse:

- Fixed groups, reliable processes: all group members should receive the multicast message. This scenario is easy to implement on top of **unreliable** multicast through either positive or negative acknowledgements
- Unreliable processes: all non-faulty group members should receive the message, but there must be an agreement about who is **inside the group**

A basic approach for non-faulty processes is described in the following picture:

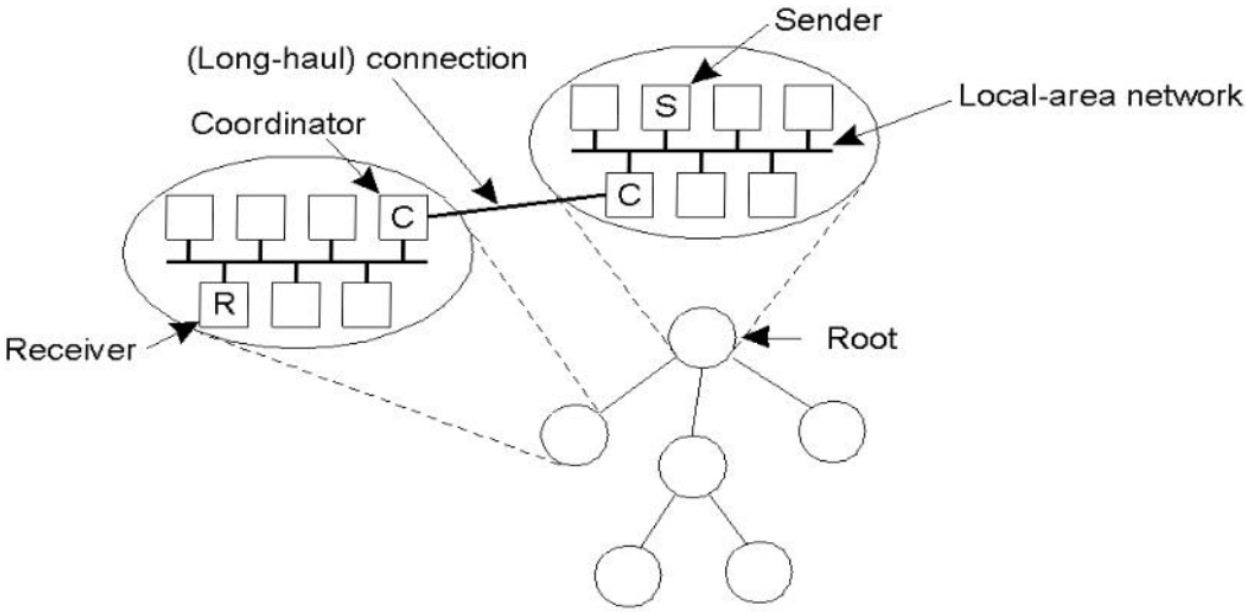


Sending acks back to the sender is not efficient though, especially in very big groups. A better approach is using multicast nacks:



This method is called **non-hierarchical feedback control**: every receiver that does not receive a packet starts a timeout timer; when this expires, the receiver sends out a nack in broadcast, so that the sender can retransmit and other receivers who did not receive the same packet don't have to send another nack, which could lead to nack implosion.

Another method that can be used is **hierarchical feedback control**, in which a *leader* is designated to coordinate the feedback of the whole group:



The group coordinator can adopt any strategy within its group. In addition, it can request retransmissions to its parent coordinator.

A coordinator can remove a message from its buffer if it has received an ack from either all the receivers in its group or all of its child coordinators.

Between groups, communication is accomplished through P2P.

The problem with hierarchical feedback control is that the hierarchy tree has to be constructed and maintained for it to work.

Faulty processes

Things get much harder if processes can fail or join and leave the groups during communication.

What is usually needed in this case is that a message needs to be delivered to either all the members of a group or to none. Furthermore, the order of messages needs to be the same at all receivers.

This is known as the **atomic multicast problem**.

Close synchrony

Ideally, we would like that:

- Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order
- A multicast to a process group is delivered to its full membership. The send and delivery events should be considered to occur as a single, instantaneous event

Unfortunately, close synchrony cannot be achieved in the presence of failures.

Virtual synchrony

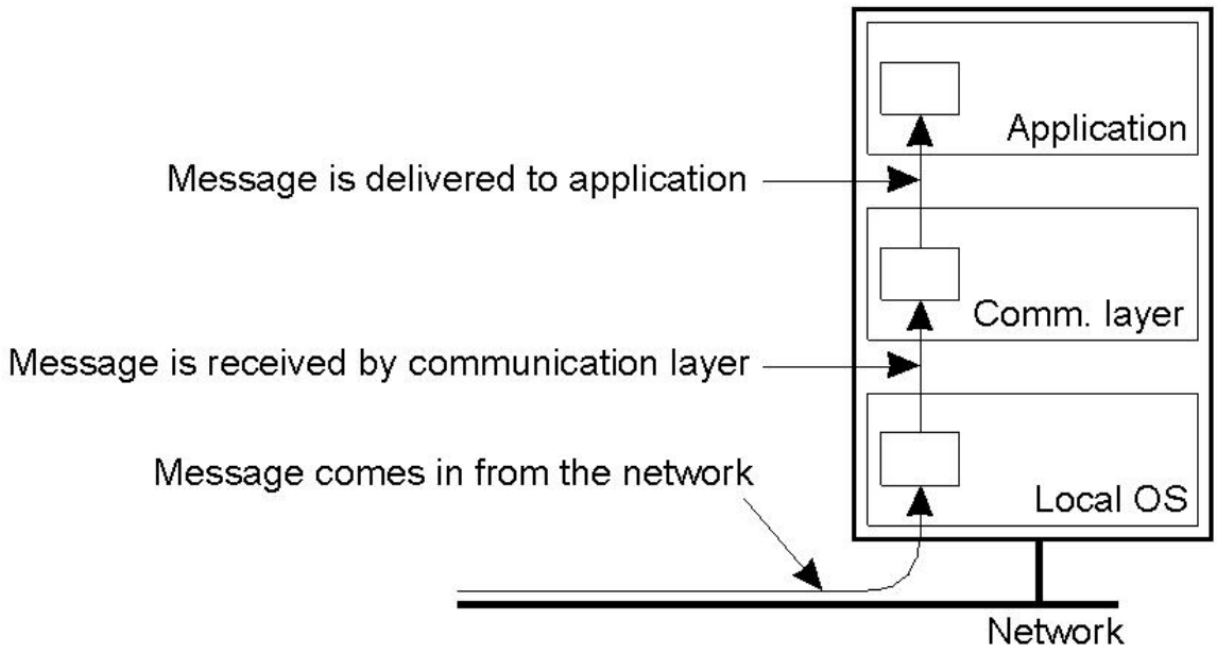
A mechanism to detect failures is needed, but even if we can detect failures correctly, we cannot know whether a failed process has received and processed a message.

Our new, weaker model becomes:

- Crashed processes are purged from the group and have to join again, reconciling their state

- Messages from a correct process are processed by all correct processes
- Messages from a failing process are processed either by all correct members or by none
- Only relevant messages are received in a specific order

To our multicast primitive, it is useful to adopt the following model which distinguishes between receiving and delivering a message: messages received by the communication layer are buffered and later delivered, when some condition holds:

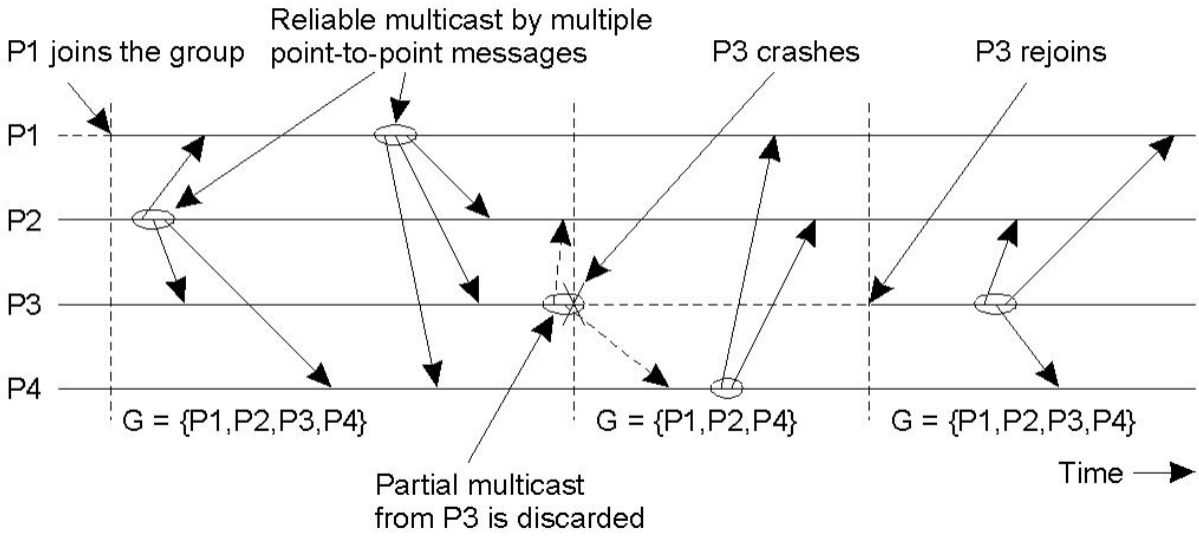


To implement virtual synchrony, we must define a **group view**:

> **Group view**
>
> A group view is the set of processes to which a message should be delivered as seen by the sender at sending time

The minimal ordering requirement is that group view changes must be delivered in a consistent order with respect to other multicasts and with respect to each other.

This, together with the previous requirements, leads to a form of reliable multicast which is said to be **virtually synchronous**.



In virtual synchrony, we say that a **view change** occurs when a process joins or leaves the group, possibly crashing.

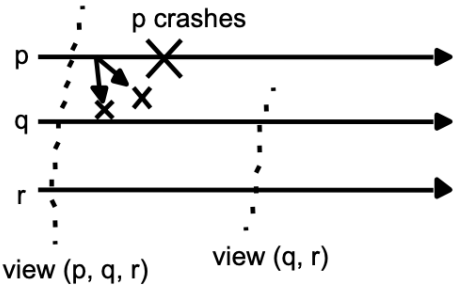
All multicast must take place between view changes:

- We can see view changes as another form of multicast messages
- We must guarantee that messages are always delivered before or after a view change
- If the view change is the result of the sender of a message m leaving, the message is either delivered to all group members before the view change is announced or it is dropped

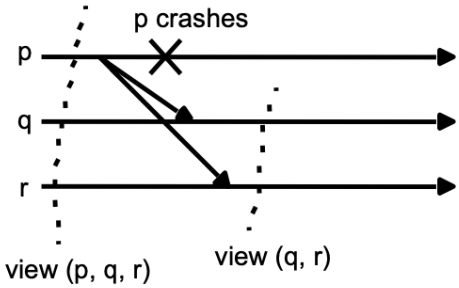
Multicasts take place in **epochs** separated by group membership changes.

Let's summarise what is allowed and what is not in virtual synchrony:

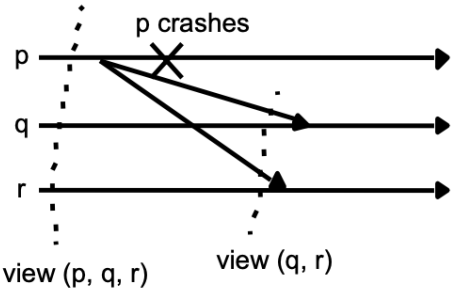
a (allowed).



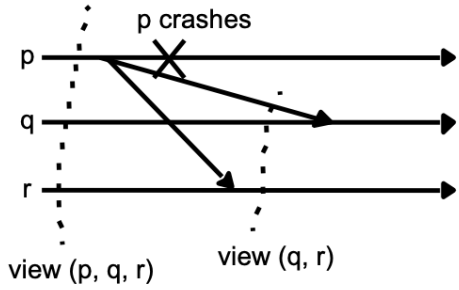
b (allowed).



c (disallowed).



d (disallowed).



Retaining the virtual synchrony property, we can identify different orderings for the multicast messages:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts

In addition, the above orderings can be combined with a **total** ordering requirement: whatever ordering is chosen, messages must be delivered to every group member in the same order. Virtual synchrony includes this property in its own definition.

We can now redefine atomic multicast:

> **Atomic multicast**
>
> Atomic multicast is defined as a virtually synchronous reliable multicast offering totally-ordered delivery of messages

We can now summarise all kinds of multicast with this table:

Multicast	Basic message ordering	Total-ordered delivery
Reliable multicast	None	No

Multicast	Message ordering	Total-ordered delivery
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

An implementation of virtual synchrony

Virtual synchrony was implemented in ISIS, a fault tolerant distributed system, making use of reliable and FIFO P2P channels.

In practice, although each transmission is guaranteed to succeed, there are no guarantees that all group members receive it: the sender may fail before completing its job.

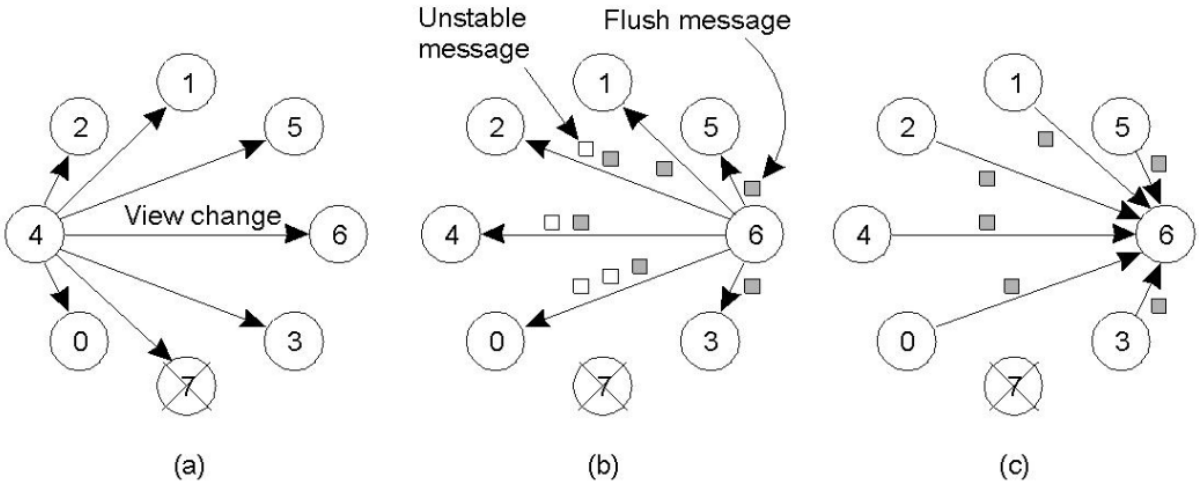
ISIS must make sure that messages sent to a group view are all delivered before the view changes: every process keeps a message m until it is sure that all the others have received it. Once this happens, m is said to be **stable**. We assume that processes are notified when messages become stable and that they keep them in a buffer until that time.

The basic idea behind ISIS is the following:

- We assume that processes are notified of view changes by some component
- When a process receives a view change message, no new messages should be sent until the new view is installed; instead, all pending unstable messages are multicasted, marked as stable and then flushed through a multicast flush message
- Eventually, all the non-faulty members of the old view will receive the view change and do the same
- Each process installs the new view as soon as it has received a flush message from every other process in the view. Now, it can restart sending new messages

If we assume that the group membership does not change during the execution of the protocol above, we have that, in the end, all non-faulty members of the old view receive the same set of messages before installing the new view.

Let's take a look at an example:



1. Process 4 notices that process 7 has crashed and sends a view change (picture a)
2. Process 6 sends out all its unstable messages, followed by a flush message (picture b)
3. Process 6 installs the new view when it has received a flush message from everyone else (picture c)

Recovery techniques

When processes resume working after a failure, they have to be taken back to a correct state. This is called **recovery**.

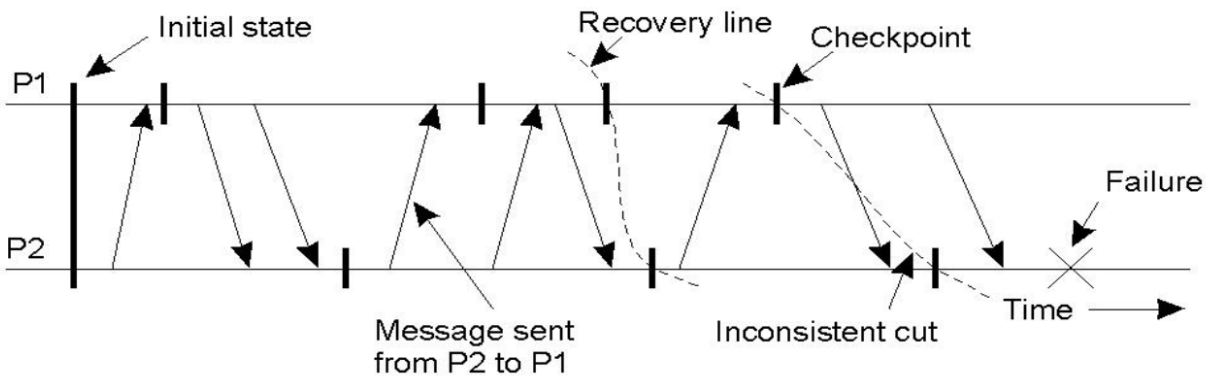
There are two main types of recovery:

- **Backward recovery**: the system is brought back to a previously saved correct state
- **Forward recovery**: the system is brought into a new correct state from which execution can be resumed

Recovering a previous state is only possible if that state can be **retrieved**. **Checkpointing** consists in periodically saving the distributed state of the system to stable storage. This is a very expensive operation though.

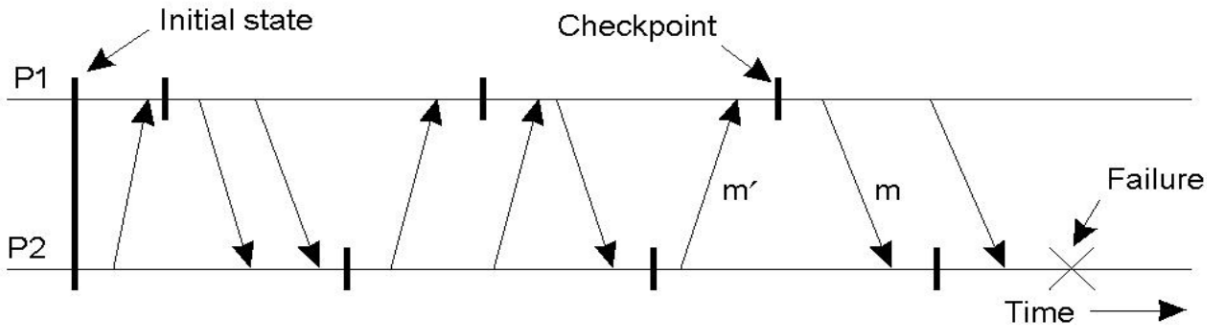
A simpler way to record the state of a distributed system is through **logging**: events and messages are recorded to stable storage so that they can be replayed when recovering.

In order to recover the state of a distributed system, we need to find a consistent "cut", called **recovery line**:



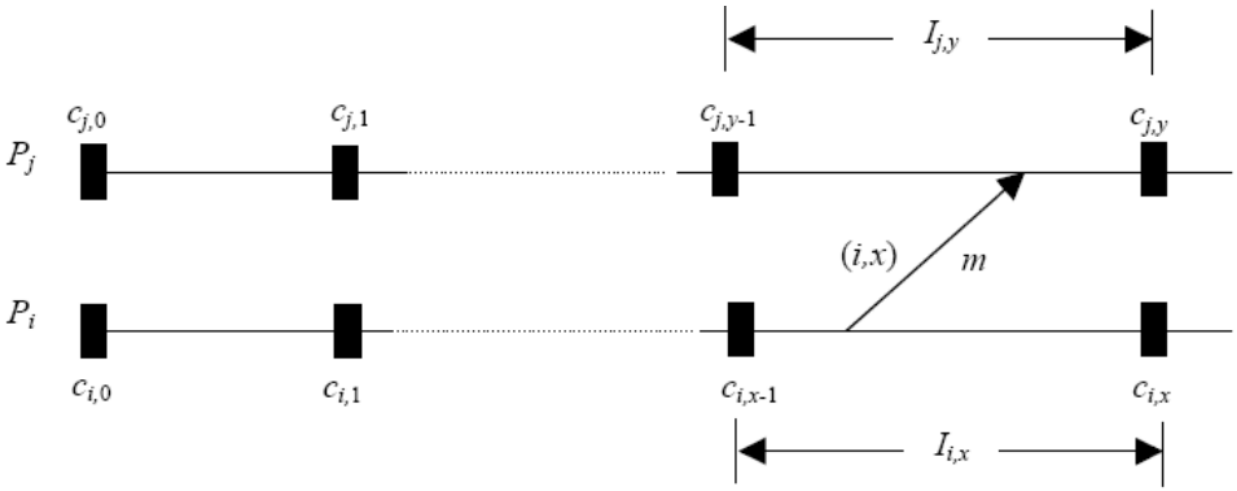
Independent checkpointing

The most trivial solution is to have each process independently record its own state. We may need to roll back to previous checkpoints until we find a set of checkpoints which result in a consistent cut. This is called **domino effect**.



Independent checkpointing is also not trivial to implement, since the interval between two checkpoints needs to be tagged and each message exchanged by the process must be recorded with a reference to the interval.

Let $c_{i,x}$ be the x-th checkpoint taken by process P_i and $I_{i,x}$ be the interval between checkpoints $c_{i,x-1}$ and $c_{i,x}$. When a process sends a message to another, it piggybacks information about its current checkpoint interval. The receiver uses this information to record that its own interval is dependent on the sender's.

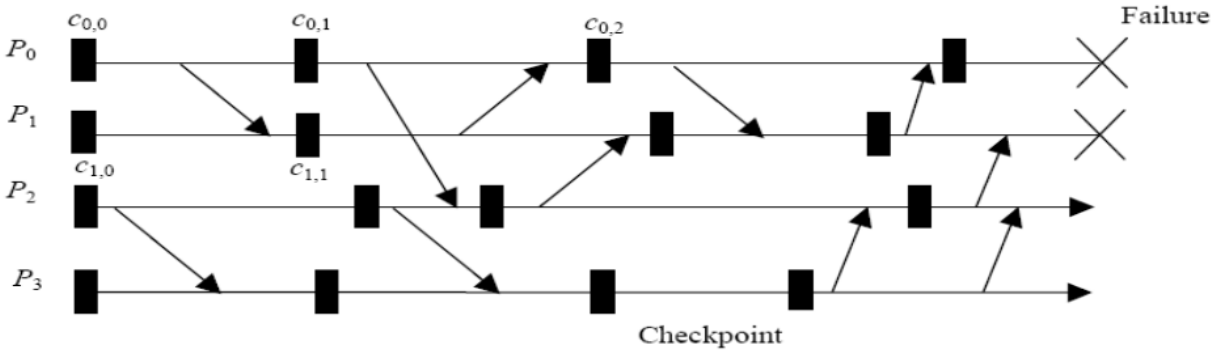


When a failure occurs, the recovering process broadcasts a dependency request to collect dependency information. At this point, all processes stop and send their information and the recovering process computes the recovery line and sends the corresponding rollback request.

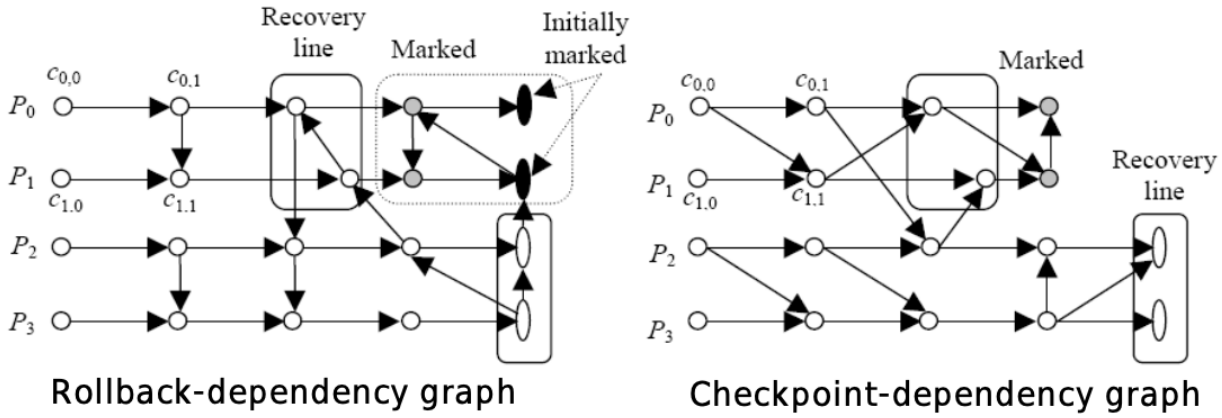
The recovery line can be computed using two approaches:

- The **rollback dependency graph**
- The **checkpoint graph**

If we consider the following scenario:



Then the recovery line found by both algorithms is:



In the rollback-dependency graph, we draw an arrow between two checkpoints $c_{i,x}$ and $c_{j,y}$ if either:

- $i = j$ and $y = x + 1$
- $i \neq j$ and a message is sent from $I_{i,x}$ to $I_{j,y}$

The recovery line is computed by marking the nodes corresponding to the failure states and then marking all those which are reachable from one of them.

An equivalent approach can be used on the checkpoint graph: take the latest checkpoints that do not have dependencies among them.

Coordinated checkpointing

The complexity of the previous algorithm suggests that independent checkpointing is not so convenient. The solution to this problem is to **coordinate checkpoints** so that no useless checkpoints are ever taken.

A simple implementation of this is the following:

1. The coordinator sends `CHKP-REQ`
2. Receivers take a checkpoint and queue other outgoing messages
3. Once the operation is done, the receivers respond to the coordinator with an acknowledgement
4. When all operations complete, the coordinator sends out a `CHKP-DONE` message

Another improvement of this technique consists in **incremental snapshots**, which only request checkpoints to processes that depend on recovery of the coordinator, i.e. those processes that have received a message causally dependent from one sent by the coordinator after the last checkpoint.

Furthermore, a global snapshot algorithm can be used to take a coordinated checkpoint:

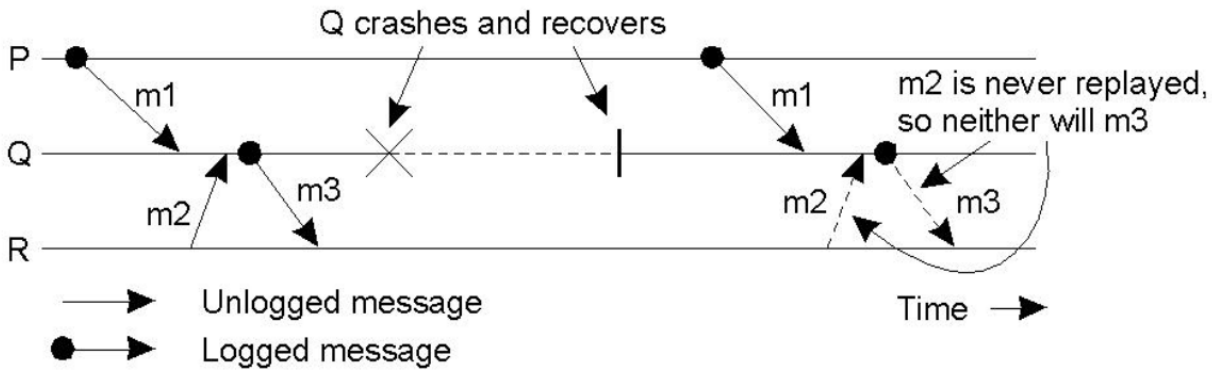
- **Advantage**: processes are not blocked while the checkpoint is being taken
- **Drawback**: increased complexity of the algorithm

Another alternative which tries to share the benefits of independent and coordinated checkpointing is **communication-induced checkpointing**, in which processes take checkpoints independently but piggyback information on messages, in order to allow the other processes to determine whether they should also take a checkpoint.

Logging schemes

Thanks to logging, we can start from a checkpoint and replay the actions that are stored in a log file. This method works if the system is *piecewise deterministic*: execution proceeds deterministically in the time that two messages are received.

Logging must be done carefully, as this picture suggests:



In logs, each message header contains all necessary information to replay the messages.

A message is said to be **stable** if it can no longer be lost (i.e. it has been written to stable storage).

For each **unstable** message m, we define:

- DEP(m): processes that depend on the delivery of m, i.e. those to which m has been delivered or to which m', which depends on m, has been delivered
- COPY(m): processes that have a copy of m, but not yet in stable storage. These processes should hand over a copy of m that could be used to replay it

eventually

Given these definitions, we can characterise an orphan process in the following way:

> Let Q be one of the surviving processes after one or more crashes; Q is an orphan if there exists a message m such that Q is in $\text{DEP}(m)$ and all processes in $\text{COPY}(m)$ have crashed

If each process in $\text{COPY}(m)$ has crashed, then no surviving processes must be left in $\text{DEP}(m)$. This can be obtained by having all processes in $\text{DEP}(m)$ also be in $\text{COPY}(m)$. Furthermore, when a process becomes dependent on a copy of m, it keeps m.

Pessimistic vs. optimistic logging

Let's differentiate between two logging techniques:

Type of logging	Description	Characteristics
Pessimistic	Ensure that any unstable message m is delivered to at most one process	The receiver will always be in $\text{COPY}(m)$, unless it discards the message; the receiver cannot send any other message until it writes m to stable storage; if communication is assumed to be reliable, then logging should ideally be performed **before message delivering**.
Optimistic	Messages are logged asynchronously, with the assumption that they will be logged before any fault occurs	If all processes in $\text{COPY}(m)$ crash, then all processes in $\text{DEP}(m)$ are rolled back to a state where they are no longer in $\text{DEP}(m)$; this is similar to the technique used in independent checkpointing.

Pessimistic logging protocols ensure that **no orphan processes are ever created**, while optimistic logging protocols allow orphans to be created and **force them to roll back until they are no longer orphans**.

A variant of pessimistic logging requires messages to be **logged by the sender**.

Logging was originally designed to allow the use of independent checkpointing. However, it has been shown that combining coordinated checkpointing with logging yields **better performance**.